Name: Dhruv Singal

Q1 A) Quantity to be optimized = Change (i)

this represents the possibility of creating change of i value from the coin supply of denominations $d_1 < d_2 < \dots < d_n$

B) Recurrence Relation:

Change(i) = min{dj, where dj ≤ i, and Change($i - d_j$) is ≥ 0 (≥0 means possible
−1 means impossible)
} // the min is there because how
// the loop is working in code

C) Base case: Change (0) = 0,
assuming there is no coin of value 0.
Change (k) = −1 for 1 ≤ k ≤ V ← final value

D) Final Solution: Change (V)

E)
```
int[] denom; // sorted denominations increasing.
int[] table = new int[V+1]
//initialization
table[0] = 0        // 0 value needs 0 coins.
for(int i=1; i<=v; i++)
{
    table[i] = -1;  // -1 means not possible
}
// fill table
for(int i=1, i<=v; i++)
{
    int coinVal = table[i]    // -1
    int j= 0 1;
while( coinVal <0  &&  denom[j] <=i && j<n )
{    coinVal = table [i-denom [j]]
     j++ ;
 }
 j--;   // counteract last j++
// j now points to coin that was picked up
// iff coinVal ≥0
if (coinVal >=0) {
    table [i] = denom[j] ;
}
}
```

Q2: A: Quantity to to optimized = MaxProfit(m_i) where, m_i is the i'th potential restaurant location. MaxProfit(m_i) signifies the maximum profit to be made by placing a restaurant up to and including the i'th location.

B: Recurrence Relation = MaxProfit(m_i) = max{p_i + MaxProfit(m_a), MaxProfit(m_b)} where p_i is the profit to be made by placing a restaurant at m_i. m_a is the closest location that is at least k miles before m_i and m_b is the closest location before m_i but is not m_i itself.

C: Bases cases: MaxProfit(m_0) = 0. Assuming that m_0 is an imaginary location that is k away from m_1 and no restaurant can be placed there and yields 0 profit.

D: Final solution = MaxProfit(m_n) where n = number of locations available to put a restaurant at.

E: Pseudo code for table filling:

```
// variables
int[] profits; //array of profits for each location starting at p_1 and in order of location
int[] locations; // array of locations in order of distance along Highway 1 starting with m_1
int n = locations.length+1; // number of locations including m_0.
int[] table = new int[locations.length+1]; // to store m_0 as well
// initialization
table[0]=0; // no restaurant can be put at m_0.
// table fillup
for(int i=1;i<n;i++){
        // find locations compatible
        int locationNear=0; // location just before
        int locationKFar=0; // location at least k miles before
        // j goes through locations in location array that starts with m_1 not m_0
        for(int j=i-1; j>=0;j--){
                if(locations[i]-locations[j]>=k){
                        locationKFar=j+1;
                }
                if(locations[i]-locations[j] != 0){
                        locationNear=j+1;
                }
                if(locationNear>-1 && locationFar>-1){
                        break;
                }
        }
        table[i]=Math.max(profits[i-1]+table[locationFar], table[locationNear]);
}
```

Q3: You are going on a long trip. You start on the road at mile post 0. Along the way there are n hotels, at mile posts a1 < a2 < … < an, where each ai is measured from the starting point. The only places you can stop are at these hotels, but you can choose which of the hotels you stop at. You must stop at the final hotel (at distance an), which is your destination. You would ideally like to travel 300 miles a day, but this may not be possible (depending on the spacing of the hotels). If you travel x miles during a day, the penalty for that day is (300 x)2 . You want to plan your trip to minimize the total penalty-that is, the sum, over all travel days, of the daily penalties. Give an efficient algorithm that determines the optimal sequence of hotels at which to stop.

A. Quantity to be optimized: minPenalty(a_i), which the penalty on reaching hotel a_i.
B. Recurrence Relation: minPenalty(a_i) = min(minPenalty(a_j)+(300-x_j)^2), where a_j is a hotel before a_i, and x_j is the distance between a_j and a_i.
C. Base cases: minPenalty(a_0)=0. Penalty for imaginary hotel at the start of the trip is 0.
D. Final solution: minPenalty(a_n) is the minimum penalty incurred travelling to a_n.
E. Pseudocode for table filling:

```
// variables
int[] hotels; //n long array with hotels in ascending order of distance along trip
int numHotels=hotels.length+1; // number of hotels in table including $a_0$
int[] table = new int[numHotels] //creating space for $a_0$.
// initialization
table[0]=0;
for(int i=1;i<numHotels;i++){
    int min=INTEGER_MAX;
    // this for loop finds the minimum penalty to get from a previous hotel to $a_i$.
    for(int j=i-1;j>=0;j--){
        if(table[j] + (300-(hotels[i]-hotels[j]))$^2$ <min){
            min=table[j] + (300-(hotels[i]-hotels[j]))$^2$;
        }
    }
    table[i]=min;
}
```

Q4:

a) Legal patterns for a column:
   0000, 1000, 0100, 0010, 0001, 1010, 0101, 1001 // 1 denotes presence of pebble.
b) Compatibility list:
   i) 0000: 0000, 1000, 0100, 0010, 0001, 1010, 0101, 1001
   ii) 1000: 0000, 0100, 0010, 0001, 0101
   iii) 0100: 0000, 1000, 0010, 0001, 1010, 1001
   iv) 0010: 0000, 1000, 0100, 0001, 0101, 1001
   v) 0001: 0000, 1000, 0100, 0010, 1010
   vi) 1010: 0000, 0101, 0100, 0001
   vii) 0101: 0000, 1010, 1000, 0010
   viii) 1001: 0000, 0100, 0010

A. Quantity to be optimized: MaxSum(k). MaxSum(k) is the sum of the integers accrued upto and including the k^{th} column of the checkerboard.

B. Recurrence Relation: MaxSum(k)=max(MaxSum(k-1)+V_k), where MaxSum(k-1) is the maximum accrued value for k-1 columns, and V_k is the value of the k^{th} column(ie, the value gained by placing pebbles in each of the patterns compatible with the $k\text{-}1^{th}$ columns pattern.)

C. Bases cases: Assume an imaginary 0_{th} column with no pebbles. Therefore, MaxSum(0)=0.

D. Final solution: MaxSum(n), where n is the number of columns in the checkerboard.

E. Pseudocode for table filling

```
// variables
int[][] checkerBoard; // grid of dimensions 4xn with integer values on the boxes
Hashtable<Integer,Integer[]> patterns;
/* hashtable of patterns, with the key being the value of the binary number formed by the pattern
eg 0101 is 5, and the value being the pattern itself. Therefore <key,value> = <5,[0,1,0,1]>
*/
Hashtable<Integer,Integer[]> compatible;
/* hashtable of patterns, with the key being the value of the binary number formed by the
patterns eg 1000 is 8, and the value being a list of keys for hashtable patterns that are
compatible with the pattern with that key.
*/

int[] table = new int[n+1]; // to store imaginary base case for the 0th column
// initialization
table[0]=0;
int prevPattern=0; //stores the pattern of the previous array. starts as 0 to handle the 0th column
// later it is the value of the binary string that represents the pattern
// fill er up
```

```
for(int i=1;i<=n;i++){
        int max=0;
        for(int j in compatible[prevPattern]){
        // for loop goes through all the keys of all patterns compatible with previous column
                if(dotProduct(checkerBoard[][i-1], patterns[j]) > max){
                        max=dotProduct(checkerBoard[][i-1], patterns[j]);
                }
                // this block went through all the compatible patterns and checked found the
        maximum dot product of the pattern, eg [0,1,0,1] with the i-1 column of the
        checkerboard.
                // i-1 because the checkerboard's first column is at index 0, not 1.
        }
        // set the value of the i'th column in the table = max+value of the i-1th column
        table[i]=max+table[i-1];
}
```